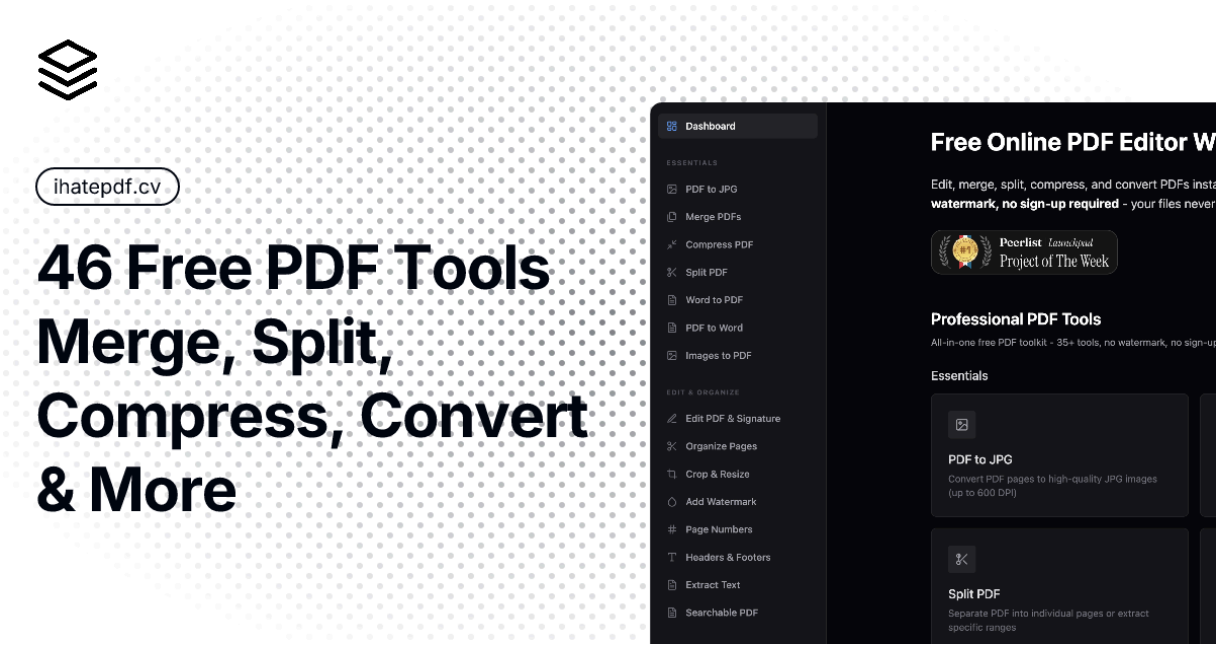
Building a Client-Side PDF Engine — How ihatepdf Works

ihatepdf.cv/technical-blog

ihatepdf June 1, 2024



01

The Challenge

Processing PDFs in the browser is notoriously difficult. Traditional web apps upload files to servers, process them remotely, and return results. This approach has critical flaws for privacy-sensitive documents: contracts, medical records, financial statements.

The core problem — PDFs are complex binary formats requiring significant RAM. A 50 MB PDF with images needs 200–300 MB to process. Multiply across concurrent users and server costs skyrocket. More importantly, sensitive documents shouldn't leave your device at all.

- Privacy risk — files travel over the network
- Upload/download bandwidth waste
- Server costs and scaling problems
- Network latency on every operation
- Single point of failure for breaches

- Complete privacy — bytes never leave device
- Instant — no network round-trip

- Zero infrastructure cost
- Works fully offline after first load
- No server = no server breach possible

JavaScript wasn't designed for heavy binary processing. A single PDF operation can block the main thread, freeze the UI, or crash the tab if memory isn't carefully managed. This is why most "free" tools either cap file sizes severely or require paid subscriptions.

02

Technical Architecture

Core Libraries

Two WebAssembly-powered engines handle all processing. Both run entirely inside the browser tab — no native installation, no server dependency.

PDF manipulation engine. Merges, splits, adds pages, edits metadata. Works at the PDF structure level — copies pages without re-rendering, making merges instant.

Handles PDF internals: form fields, annotations, encryption — that most libraries skip entirely.

Mozilla's PDF rendering engine — the same one Firefox uses natively. Converts pages to Canvas for pixel-perfect previews and high-DPI image export.

Worker architecture offloads heavy parsing to a separate thread, keeping the UI thread responsive.

Storage Strategy

Three-tier storage solves different problems. The key rule: keep data as ArrayBuffer as long as possible — converting to strings increases memory 3–5×.

javascript

```
// ── Tier 1: IndexedDB — large binary files ──────────────────
const dbSet = async (key, val) => {
  const db = await initDB();
  return new Promise((resolve, reject) => {
    const tx = db.transaction('ihatepdf-store', 'readwrite');
    tx.objectStore('ihatepdf-store').put(val, key);
    tx.oncomplete = () => resolve();
    tx.onerror   = () => reject(tx.error);
  });
```

```
};

// ── Tier 2: localStorage — metadata only (no file bytes) ─
localStorage.setItem('ihatepdf_history', JSON.stringify(history));

// ── Tier 3: RAM — active processing (volatile) ──────────
const pdfDoc = await PDFDocument.load(arrayBuffer);
```

localStorage has a 5–10 MB limit and stores data as strings, doubling memory usage. IndexedDB stores binary data natively and can handle gigabytes. The trade-off: async API requiring Promise handling everywhere — but the capacity difference is non-negotiable for large PDFs.

03

Memory Management

The app dynamically adjusts limits based on device type and available memory. A 4 GB RAM phone typically has 1–1.5 GB available for browser tabs. A single 100 MB PDF can consume 300–400 MB during processing — 3–4× overhead for rendering.

javascript

```
const getDeviceCapabilities = () => {
  const isMobile   = /Android|iPhone/i.test(navigator.userAgent);
  const deviceMem  = navigator.deviceMemory || 4; // GB (not in Safari)

  if (isMobile && screen.width < 768) {
    return {
      maxFileSize:       50  * 1024 * 1024,  // 50 MB
      maxDPI:            300,
      maxPagesPerBatch:  10,
      warningThreshold:  30  * 1024 * 1024,
    };
  }
  if (deviceMem < 4) {
    return { maxFileSize: 100 * 1024 * 1024, maxDPI: 450, maxPagesPerBatch: 30
};
  }
  return {
    maxFileSize:       150 * 1024 * 1024,  // 150 MB — high-end desktop
    maxDPI:            600,
    maxPagesPerBatch:  50,
  };
};
```

Device	Max File	Max DPI	Batch Size	Scale
Smartphone	50 MB	300	10 pages	0.6 – 0.8×
Tablet	75 MB	450	25 pages	0.8 – 1.2×
Desktop	150 MB	600	50 pages	1.0 – 2.0×

Memory Estimation Algorithm

Before any heavy operation, the app estimates RAM consumption. Memory scales quadratically with scale — 2× scale means 4× memory usage — and PNG needs ~1.5× more than JPEG.

javascript

```
const estimateMemoryUsage = (fileSize, pageCount, scale, format) => {
  const basePerPage    = 5 * 1024 * 1024;       // ~5 MB at scale 1.0
  const scaleFactor    = Math.pow(scale, 2);     // quadratic growth
  const fmtMultiplier  = format === 'png' ? 1.5 : 1.0;
  const estimated      = pageCount * basePerPage * scaleFactor *
fmtMultiplier;
  return { estimated, withSafety: estimated * 1.5 }; // 50% safety margin
};

// Warn user before starting if this will exhaust device memory
if (estimate.withSafety > (navigator.deviceMemory || 4) * 1e9 * 0.5) {
  alert('⚠ Reduce DPI or page count before continuing.');
}
```

Canvas Memory — The Hidden Cost

Each canvas element allocates GPU texture memory in addition to RAM. A 4000×6000 canvas uses 96 MB of RAM *plus* 96 MB of VRAM. On devices with shared memory (most mobile), that's 192 MB gone per page.

javascript

```
// × BAD — memory leak, canvas stays allocated indefinitely
const canvas = document.createElement('canvas');
canvas.width  = 4000;
canvas.height = 6000;
// ... use canvas, then forget it

// ✓ GOOD — explicit GPU + RAM release
const canvas = document.createElement('canvas');
canvas.width  = 4000;
canvas.height = 6000;
```

```
const blob = await new Promise(r => canvas.toBlob(r, 'image/jpeg', 0.95));
await processBlob(blob);

canvas.width  = 0;   // ← triggers immediate GPU memory release
canvas.height = 0;
canvas = null;
```

Batch Processing

When converting 100+ pages, processing all at once crashes the browser. The solution: intelligent batching with deliberate pauses for garbage collection.

javascript

```
const processBatches = async (allPages, batchSize) => {
  const batches = [];
  for (let i = 0; i < allPages.length; i += batchSize)
    batches.push(allPages.slice(i, i + batchSize));

  for (let b = 0; b < batches.length; b++) {
    for (const page of batches[b]) {
      const canvas = await renderPage(page);
      await downloadImage(canvas);
      canvas.width = canvas.height = 0;   // free immediately
    }

    if (b < batches.length - 1) {
      // 2 s pause — Chrome GC triggers after ~1–1.5 s idle
      await new Promise(r => setTimeout(r, 2000));
      if (window.gc) window.gc(); // hint only — browser may ignore
    }
  }
};
```

04

Feature Implementation

PDF Compression — Ghostscript via WebAssembly

The compression engine uses Ghostscript compiled to WebAssembly — the same tool used by professional PDF software. Unlike simple image re-compression, Ghostscript optimizes the entire PDF structure: font subsetting, object-stream compression, and intelligent image downsampling.

javascript

```
const qualityPresets = {
  '/screen':   { dpi: 72,  jpeg: 40 },  // Screen viewing
  '/ebook':    { dpi: 150, jpeg: 60 },  // Tablets, e-readers
  '/printer':  { dpi: 300, jpeg: 80 },  // Office printing
  '/prepress': { dpi: 300, jpeg: 92 },  // Professional print
};

// Runs in a Web Worker — UI stays responsive during 10–30 s compression
const worker = new Worker('/background-worker.js');
worker.postMessage({ data: { psDataURL: blobUrl, config }, target: 'wasm' });
worker.onmessage = async (e) => {
  const response       = await fetch(e.data);
  const compressedBlob = await response.blob();
};
```

What Ghostscript actually does

Five simultaneous optimisations: (1) image downsampling via bicubic interpolation, (2) JPEG recompression at the target quality level, (3) font subsetting — trims embedded fonts to only characters used (up to 90% reduction), (4) metadata stripping, (5) stream recompression with the most efficient lossless algorithm. Text and vector graphics are entirely unaffected.

High-DPI PDF → Image Conversion

The converter supports up to 600 DPI with automatic device-aware limits, a canvas size clamp at 16,384 px, and batch mode for large documents.

javascript

```
// 72 DPI is the browser's base resolution (1 CSS px = 1 device px @ 1× zoom)
const dpiToScale = (dpi) => dpi / 72;
// 300 DPI → 4.17×   |   600 DPI → 8.33×

const getOptimalScale = (viewport, requested) => {
  const MAX = 16384; // hard browser canvas limit
  const w   = viewport.width  * requested;
  const h   = viewport.height * requested;
  if (w > MAX || h > MAX) {
    const safe = Math.min(MAX / viewport.width, MAX / viewport.height);
    return safe * 0.95; // 5% margin
  }
  return requested;
};

const renderPageToCanvas = async (page, scale) => {
  const viewport   = page.getViewport({ scale });
  const pixelRatio = Math.min(window.devicePixelRatio || 1, 2);
  const canvas     = document.createElement('canvas');

  canvas.width  = Math.floor(viewport.width  * pixelRatio);
```

```
  canvas.height = Math.floor(viewport.height * pixelRatio);

  const ctx = canvas.getContext('2d', { alpha: false, willReadFrequently:
false });
  ctx.fillStyle = 'white';
  ctx.fillRect(0, 0, canvas.width, canvas.height);
  ctx.scale(pixelRatio, pixelRatio);
  ctx.imageSmoothingEnabled = true;
  ctx.imageSmoothingQuality = 'high';

  await page.render({ canvasContext: ctx, viewport, intent: 'print'
}).promise;
  return canvas;
};
```

Safari-Compatible Downloads

Safari's security model blocks programmatic downloads. Three-method fallback: download.js → HTML5 anchor → window.open() which triggers iOS's share sheet.

javascript

```
const safariCompatibleDownload = (data, filename, mimeType) => {
  // Method 1 — download.js library
  if (typeof window.download === 'function') {
    window.download(data, filename, mimeType); return;
  }
  // Method 2 — HTML5 download attribute
  const blob = new Blob([data], { type: mimeType });
  const url  = URL.createObjectURL(blob);
  const a    = Object.assign(document.createElement('a'), {
    href: url, download: filename, style: 'display:none',
  });
  document.body.appendChild(a);
  a.click();
  setTimeout(() => {
    document.body.removeChild(a);
    URL.revokeObjectURL(url); // critical — prevents memory leak
  }, 1000);
};
```

05

AI-Powered Chat with PDF

The Chat with PDF feature combines local text extraction with Google's Gemini 2.5 Flash API. The PDF binary never reaches Google's servers — pdf.js extracts plain text locally, which is sent as the prompt context. This balances privacy (no file uploads) with AI capability.

javascript

```
// Step 1 — Extract text locally with pdf.js
const extractText = async (file) => {
  const buffer = await readFileAsArrayBuffer(file);
  const pdf    = await window.pdfjsLib.getDocument({ data: buffer }).promise;
  let text     = '';
  for (let i = 1; i <= pdf.numPages; i++) {
    const page    = await pdf.getPage(i);
    const content = await page.getTextContent();
    text += `

--- Page ${i} ---
${content.items.map(x => x.str).join(' ')}`;
  }
  return text;
};

// Step 2 — Stream response from Gemini API
const res = await fetch(
  `https://generativelanguage.googleapis.com/v1beta/models/gemini-2.5-
flash:streamGenerateContent?key=${key}&alt=sse`,
  {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({
      contents: [{ parts: [{ text: `Document:
${extractedText}

Q: ${userQuery}` }] }],
      generationConfig: { temperature: 0.7, maxOutputTokens: 2048 },
    }),
  }
);

// Step 3 — Parse SSE stream for real-time token output
const reader  = res.body.getReader();
const decoder = new TextDecoder();
while (true) {
  const { done, value } = await reader.read();
  if (done) break;
  decoder.decode(value).split('
')
    .filter(l => l.startsWith('data: '))
    .forEach(l => {
      const token = JSON.parse(l.slice(6)).candidates?.[0]?.content?.parts?.
[0]?.text;
      if (token) appendToUI(token);
    });
}
```

Handles 100-page PDFs (~50,000 tokens) in under 3 seconds. Costs 1/10th of GPT-4. Supports streaming responses so the UI feels instantaneous. Up to 10 PDFs can be loaded simultaneously with concatenated text and document separators — enabling cross-document queries like "Compare pricing between contracts A and B."

06

Adaptive Processing

The same codebase runs on a 2 GB RAM phone and a 32 GB workstation. Rather than blocking features on low-end devices, the app scales quality down automatically. A phone user still converts PDFs to images — just at 150 DPI instead of 600 DPI.

javascript

```
const memoryEstimate = estimateMemoryUsage(fileSize, pageCount, scale,
format);
const availableGB    = navigator.deviceMemory || 4;
const estimatedGB    = memoryEstimate.withSafety / (1024 ** 3);

if (estimatedGB > availableGB * 0.5) {
  const proceed = confirm([
    `⚠ This will use ~${estimatedGB.toFixed(1)} GB.`,
    `Your device has ~${availableGB} GB available.`,
    'Recommendations:',
    '  • Reduce DPI or quality',
    '  • Process fewer pages',
    '  • Use JPEG instead of PNG',
    'Continue anyway?',
  ].join('
'));
  if (!proceed) return;
}

// Auto-enable batch mode for jobs that exceed per-device page limits
if (pageCount > deviceCaps.maxPagesPerBatch) {
  await processBatches(pagesToConvert, deviceCaps.maxPagesPerBatch);
}
```

What happens without these safeguards
A user converting a 50-page PDF to 600 DPI on a phone would freeze the browser for 5+ minutes, consume 2–3 GB of RAM crashing the tab, potentially reboot the device on older phones, and lose all progress with no error message. With adaptive processing, the app either reduces quality automatically, enables batch mode, or warns the user upfront.

07

Security & Privacy

ihatepdf operates on a simple principle: we can't leak what we never see. Files never touch our servers, third-party APIs, or external services. All processing happens in the browser's sandboxed environment.

Data Flow Verification

Open DevTools → Network tab during any PDF operation. You will see zero upload requests to any external domain. Every byte of your document stays inside the browser tab.

javascript

```
// The complete data lifecycle — nothing external
FileReader.readAsArrayBuffer(file)   // → browser memory only
  → PDFDocument.load(arrayBuffer)    // → WebAssembly execution (local)
  → pdfDoc.save()                    // → new ArrayBuffer (local)
  → new Blob([bytes])                // → browser memory
  → URL.createObjectURL(blob)        // → local object URL
  → anchor.click()                   // → device storage
// No network request anywhere in this chain
```

Offline-First via Service Worker

After the first page load, all tools function without any internet connection. Enable airplane mode and process PDFs normally — the WebAssembly libraries are cached locally.

javascript

```
// sw.js — cache-first strategy
self.addEventListener('install', (e) => {
  e.waitUntil(
    caches.open('ihatepdf-v1').then((cache) => cache.addAll([
      '/',
      'https://unpkg.com/pdf-lib@1.17.1/dist/pdf-lib.min.js',
      'https://cdnjs.cloudflare.com/ajax/libs/pdf.js/3.11.174/pdf.min.js',
    ]))
  );
});

self.addEventListener('fetch', (e) => {
  e.respondWith(
    caches.match(e.request).then((cached) => cached || fetch(e.request))
  );
});
// Test: DevTools → Application → Service Workers → check "Offline" → reload
```

Three-Tier Storage Security

RAM Volatile
Active PDFs, processing buffers. Wiped on tab close. Zero persistence.

```
const pdfDoc = await PDFDocument.load(arrayBuffer);
```

IndexedDB Session
Large file buffers for resume capability. Same-origin isolated. Manual clear available.

```
await dbSet('editor_file_buffer', arrayBuffer);
```

localStorage Metadata only
Filenames, timestamps, sizes. NO file content. Non-sensitive only.

All storage is origin-isolated and client-side. Clearing your browser data removes everything completely — no cloud sync, no backup, no recovery. The trade-off for absolute privacy is that there is no undo for deleted history.